IITM Web M.Tech Program

Course ID 6002W : RL Course Project

Project Overview: In this project, you will apply reinforcement learning techniques to solve a practical inventory control problem involving 3 different products. This project is designed to test your understanding of different RL techniques, all within a realistic industrial context. You are tasked with managing the inventory of a warehouse that stores several types of products.

Problem Description: You control the daily ordering decision for 3 different products with the following constraints:

- Each day, you can order up to a maximum quantity for each product.
- Each product has a volume (space occupied per unit) associated with it.
- The total volume of inventory in the warehouse cannot exceed its capacity.
- If existing volume and ordered volume exceeds total volume, then additional products are discarded though the ordered is paid for and there is a discarding cost because the item cannot be sold.
- Orders have a fixed lead time (delay before arrival) which varies for each product.
- Demand varies each day for each product.
- You incur costs based on the inventory levels, unfulfilled demand, and ordering.

Project Objective: Your objective is to develop an ordering policy that minimizes the total cost incurred due to:

- **Holding excess inventory**, which occupies valuable space and incurs daily storage costs;
- **Stockouts**, which occur when customer demand exceeds available inventory and may lead to lost sales or penalties;
- **Ordering**, which refers to placing orders to replenish warehouse inventory (not customer orders), incurs a fixed cost each time an order is placed, regardless of the quantity ordered.

Fixed Environment Configuration: The following will the values used in the simulation of the environment:

- Number of products: 3
- Inventory capacity (volume-based): 1000 volume units
- Initial inventory: 100 units of each product
- Product volumes: product_1: 2 units/item, product_2: 3 units/item, product_3: 1.5 units/item
- Holding cost per day: ₹5.00 per unit volume of any product
- Stockout cost per item: product_1: ₹400.00, product_2: ₹500.00, product_3: ₹300.00
- Ordering cost per order: product_1: ₹80.00, product_2: ₹200.00, product_3: ₹120.00

- Discarding cost per item: product_1: ₹200.00, product_2: ₹250.00, product_3: ₹150.00
- Lead time: product_1: 3 days, product_2: 2 days, product_3: 1 day
- Simulation duration: 50 days per episode

Environment Implementation:
To guarantee that all participants work with the exact same environment dynamics, a Python class InventoryEnv will be provided. You must use this class for simulating the environment.

- **Demand Simulation & Serving Customers:** At each step(action), the environment will:
    1. **Order arrivals:** At each step(action), the environment first applies lead-time arrivals for pending orders and adds received stock to the inventory.
    2. **Enforce volume capacity:** Immediately after arrivals, if total inventory volume exceeds capacity, excess units are discarded and incur a discard penalty (per-product), mirroring real-world wastage costs.
    3. **Demand generation:** The environment then simulates or reads the day's customer demand for each product (stochastic during training, deterministic during evaluation).
    4. **Serve demand:** It deducts the lesser of available inventory and demand from stock.
    5. **Record stockouts:** Any unfulfilled demand is recorded as a stockout, incurring the specified stockout cost.
    6. **Update inventory:** Inventory levels are updated after serving demand.
- **API**: The class implements the standard Gym interface:
    - reset() → returns initial state (inventory levels, pending orders).
    - step(action) → returns (next_state, base reward, done, info), where info includes details on served units and stockouts.
    - action is a list of order quantities for each product.

Action Space: Action space is discrete and contains the following 11 possible actions that can be taken in any state (quantities to be ordered) – **{0, 10, 20, …, 100}**

Reward Function:
Your agent will receive a reward (negative cost) at each time step based on the true cost function you are minimizing:

cost = sum over products [

holding_cost[product] * inventory_level[product] +

stockout_cost * unfulfilled_demand[product] +

ordering_cost * (1 if order_quantity[product] > 0 else 0) +

discard_cost[product] * discard_quantity[product]

]

Base reward = -cost/100 (The division by 100 is ensure that the absolute values are not too big)

Episodic return = Sum of rewards over 50 days (Undiscounted)

Note: During training, you may choose a **surrogate reward** to accelerate learning. For example, you could add the following terms to the base reward function (only during training):

- **Inventory balance bonus:** Reward term which encourages keeping inventory near a target level
- **Order stability penalty:** Terms which penalizes large swings in daily order quantity by subtracting a small penalty proportional to|order_quantity_t - order_quantity_{t-1}| for stability during learning and better balance between exploration and exploitation.
- **Stockout-free reward:** Terms which grants a small positive bonus (e.g., +1) for each product with zero unfulfilled demand in a day.

Leaderboard Evaluation:
Your trained policies will be evaluated on a deterministic version of the environment using 10 fixed demand sequences, each simulating 50 days. For each sequence, the total cost (negative cumulative base reward) will be computed. Participants will be ranked on the leaderboard by their **average total cost** across all 10 episodes (lower is better):

$\text{average\_cost} = (\text{cost}_1 + \text{cost}_2 + ... + \text{cost}_{10}) / 10$

- A public leaderboard will display the top-performing policies.
- This ensures fairness and discourages overfitting to any single demand pattern.
- Submission on leaderboard should be a .py file with a function run_policy(state) which will be run using a private copy of the deterministic environment.
- The evaluator will call run_policy(state) once per simulated day (50 calls per episode) by passing the state as input.
- The function should return a list of three integers (each $\in \{0,10,\ldots,100\}$)
- Your function **must not** call env.step or env.reset; the grading system handles environment control.
- You may keep global variables, if needed, inside .py file to carry information across calls.
- Order quantities that exceed warehouse capacity will be clipped by the env, incurring the usual penalties.
- **Note: The leaderboard will be set up in a week.**

Submission Requirements:

1. **Jupyter Notebook (.ipynb)**: Your notebook should include your entire training pipeline, model architecture, hyperparameter settings, and the code to generate the submission function (run_policy). Ensure your code is well documented with Markdown explanations and inline comments to guide readers through your methodology.
2. **Project Report (.docx or .pdf)**: A Word or PDF document detailing:
    - Problem setup and environment configuration.
    - Description of the RL algorithm, feature engineering, and any reward shaping applied.

- Key observations, inferences, and lessons learned during training and evaluation.